CSC2001F

Practical Assignment 2

Data Structure Comparison

Binary Search Tree vs AVL Tree

Niceta Nduku

NDKNIC001

20 March 2019

Introduction

The objective of this report is to look at the behavior of binary search trees and AVL trees and identify the performance difference between the two data structures.

To see instructions on how to run the programs, please read the readme.txt file that is in the same directory as this document.

Object Oriented Programming Design Process

The design process looks at what classes were created and how they interact in order of dependency. It was essential to create an object that contains the values in order to make storing and searching easier. The object class is then used to build the various data structures.

PowerData

This is an immutable class that creates the object that is going to be stored in the data structures. The instant variables of PowerData are Date/Time, Power and Voltage which are extracted and set when the CSV file is read. The class has methods, getDateTime and overridden toString from the object class. getDateTime returns the Date/Time of the object.

To create a new PowerData object,

```
PowerData dataItem = new PowerData (dateTime, power, voltage).
```

This class is the same as the one that was used in Assignment 1.

AVLTree

Power AVL is a class that builds the AVL tree from the power data object. The original code was from Patrick Marais [1] and was modified to take PoweData types. It contains an inner class called BinaryTreeNode that creates nodes that store PowerData objects. Creating a new AVL tree will set the root node to null. To insert a new PowerData item, the insert method `insert (PowerData d)` should be called and a new node will be added to the tree. `height, fixHeight, rotateLeft, rotateRight, balanceFactor, balance,` are all additional methods that take BinaryTree Nodes as parameters and all work together to ensure that the AVL balanced property is maintained.

The AVLTree class also contains a find method that looks for a string by comparing all the PowerData dateTimes in the tree. `Display ()` [2], prints out all the data items from the smallest to the largest (in order traversal). The class contains comparison operation counts for insert and find methods purposes. Three methods can be used to get the number of comparison operations; `getInsertOpCount()` returns the number from invoking insert, `getFindOpCount()` from find and `getOpCount()` prints out for both find and insert.

PowerAVLApp

PowerAVLApp is a class that prints out data from an AVL tree that stores PowerData items. The class contains four methods; `getData, printDateTime, printAllDateTimes` and the main method that runs the class.

When the main method is invoked by running the application in the terminal, eg by typing

`java PowerAVLApp "<some string>"`, `getData()` with be called. getData will then create a new AVL tree updating the instance variable that is an AVL tree. The CSV file will be read and powerData items will be created and inserted to the AVL tree for each line in the file.

If a user has not typed in any input i.e. by running `java PowerAVLApp`, the method `printAllDateTimes` will be called and the method `display`() from the AVLTree class will be called. The output will be all the powerData items in the list in order of date/time values. If a user has put in some string input, there will first be a check to see if it a file. If it is indeed a file (assuming that the file contains date time keys), for each key in the file, the `find` method in AVLTree is called and the number of operations for each iteration is printed out. First, the find operations, then the insert operations. If the input string is not a file, the method `printDateTime` is called with the input string as the parameter. A matching powerData item and the operations will be printed if found, otherwise, a message will be printed.

BinarySearchTree

BinarySearchTree is the second data structure that stores PowerData items. The original code was from Patrick Marais [1] and was modified to take PoweData types. It contains the general `insert` and `find` (search) methods of a binary tree. The BinarySearchTree class and AVLTree classes work similarly. However, Binary search tree does not have the methods that work together to keep the tree balanced. Operations counts for insert and find are also stored in this class, with similar methods as AVLTree to print them out.

PowerBSTApp

This class has the same methods as PowerAVLApp. However, instead of an AVLTree, it creates a new BinarySearchTree. All methods work the same.

Experiment

The aim of the experiment was to compare the operations of the two data structure types, Binary search tree, and AVL trees. This was done using a bash script that automated the process.

A subset of N date/time keys was created from cleaned_data.csv. The keys were then stored in a text file and each application was run with the file. The output from running the applications was then redirected to a text file. This was done for values of N from 1 to 500. Each application had its own script in order to run the tests simultaneously.

Generated graphs for both output text files using python. The graphs show the number of operations for the Best Case, Worst Case and Average Case for insert and find methods for each application

For the part 6, I duplicated the Power apps and changed them to take in the same csv file but with sorted data. Ran the applications for 500 keys and calculated the best worst and average cases for each application in Jupyter using python.

Results and Analysis

Part 2 and 4

Date/Time: 16/12/2006/19:51:00
Output:
Date/Time: 16/12/2006/19:51:00 3.388 233.220
Insert Operations: 3831
Find Operations: 8
Date/Time: 16/12/2006/21:39:00
Output:
Date/Time: 16/12/2006/21:39:00 3.302 236.280
Insert Operations: 3831
Find Operations: 16
Date/Time: 16/12/2006/17:43:00
Output:
Date/Time: 16/12/2006/17:43:00 3.728 235.840
Insert Operations: 3831
Find Operations: 20

Table 1: Results for PowerAVLApp Test with 3 known date/times

Unknown Case: 16/12/2007/17:43:00
Output:
Date/Time not found
Insert Operations: 3831
Find Operations: 9

Table 2: Results for PowerAVLApp Test with an unknown date/time

No parameters
Output:
16/12/2006/17:24:00 4.216 234.840
16/12/2006/17:25:00 5.360 233.630
16/12/2006/17:26:00 5.374 233.290
16/12/2006/17:27:00 5.388 233.740
16/12/2006/17:28:00 3.666 235.680
16/12/2006/17:29:00 3.520 235.020
16/12/2006/17:30:00 3.702 235.090
16/12/2006/17:31:00 3.700 235.220
16/12/2006/17:32:00 3.668 233.990
16/12/2006/17:33:00 3.662 233.860
17/12/2006/01:36:00 3.746 240.360
17/12/2006/01:37:00 3.944 239.790
17/12/2006/01:38:00 3.680 239.550
17/12/2006/01:39:00 1.670 242.210
17/12/2006/01:40:00 3.214 241.920
17/12/2006/01:41:00 4.500 240.420
17/12/2006/01:42:00 3.800 241.780

17/12/2006/01:43:00 2.664 243.310
Insert Operations: 3831
Find Operations: 0

Table 3: Top 10 results for PowerAVLApp Test with no parameters

Date/Time: 16/12/2006/19:51:00
Output:
Date/Time: 16/12/2006/19:51:00 3.388 233.220
Insert Operations: 4546
Find Operations: 2
Date/Time: 16/12/2006/21:39:00
Output:
Date/Time: 16/12/2006/21:39:00 3.302 236.280
Insert Operations: 4546
Find Operations: 26
Date/Time: 16/12/2006/17:43:00
Output:
Date/Time: 16/12/2006/17:43:00 3.728 235.840
Insert Operations: 4546
Find Operations: 24

Table 4: Results for PowerBSTApp Test with 3 known date/times

Unknown Case: 16/12/2007/17:43:00
Output:
Date/Time not found
Insert Operations: 4546
Find Operations: 15

Table 5: Results for PowerBSTApp Test with an unknown date/time

No parameters
Output:
16/12/2006/17:24:00 4.216 234.840
16/12/2006/17:25:00 5.360 233.630
16/12/2006/17:26:00 5.374 233.290
16/12/2006/17:27:00 5.388 233.740
16/12/2006/17:28:00 3.666 235.680
16/12/2006/17:29:00 3.520 235.020
16/12/2006/17:30:00 3.702 235.090
16/12/2006/17:31:00 3.700 235.220
16/12/2006/17:32:00 3.668 233.990
16/12/2006/17:33:00 3.662 233.860
17/12/2006/01:36:00 3.746 240.360
17/12/2006/01:37:00 3.944 239.790
17/12/2006/01:38:00 3.680 239.550
17/12/2006/01:39:00 1.670 242.210

17/12/2006/01:40:00 3.214 241.920
17/12/2006/01:41:00 4.500 240.420
17/12/2006/01:42:00 3.800 241.780
17/12/2006/01:43:00 2.664 243.310
Insert Operations: 4546
Find Operations: 0

Table 6: Top 10 results for PowerBSTApp Test with no parameters

Part 5

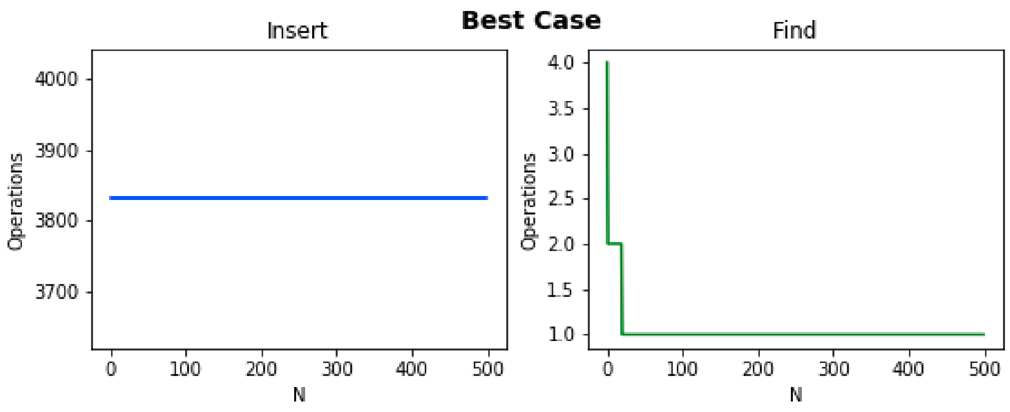

Figure 1: Best case for find and insert operation count as the data set increases in PowerAVLTree

The above graphs above show the best case as the size of the subset increased. Insert is constant because for each N, the application was run and the AVL tree was built once. The best case for find starts at four and ends at 1. This is so because of the AVL balanced property. The first item to be searched is not the first item in the tree even though when the tree is being built it was the first item to be inserted. As the subset grows the items being searched increase and hence the probability for the first item to be in the set increases.

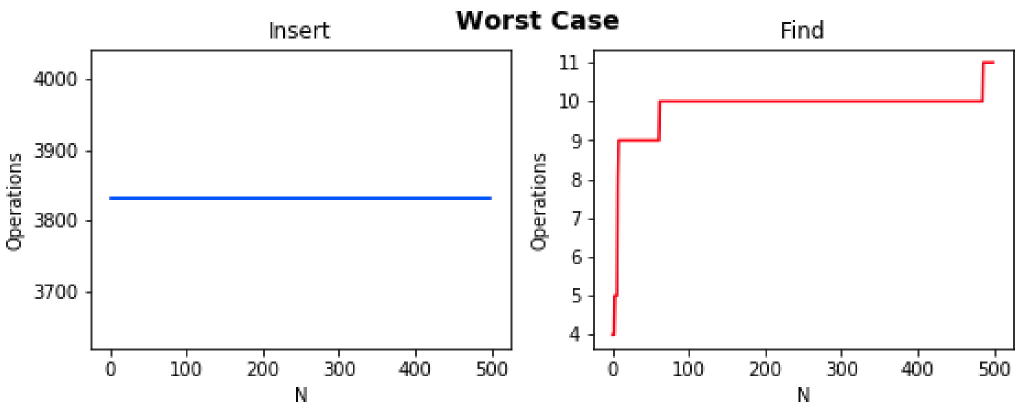

Figure 2: Worst case for find and insert operations for PowerAVLTree

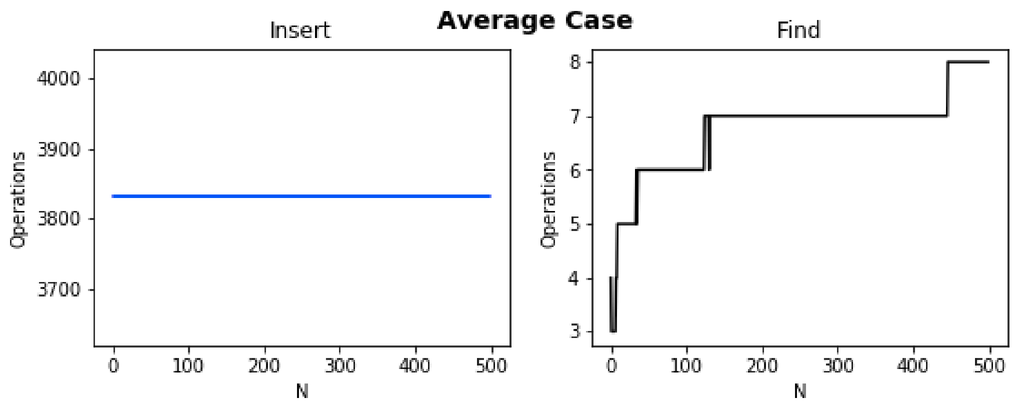

Figure 3: Average case for find and insert operations PowerAVLTree

The worst case and average case have a logarithmic pattern. This is because as the size of the subset increases the number of operations that are needed for search increase at a smaller rate. The average case graph has the above pattern because the first N value had four operations and therefore the average is at 4. It then drops for the next few N values then slowly starts to increase for greater values of N.

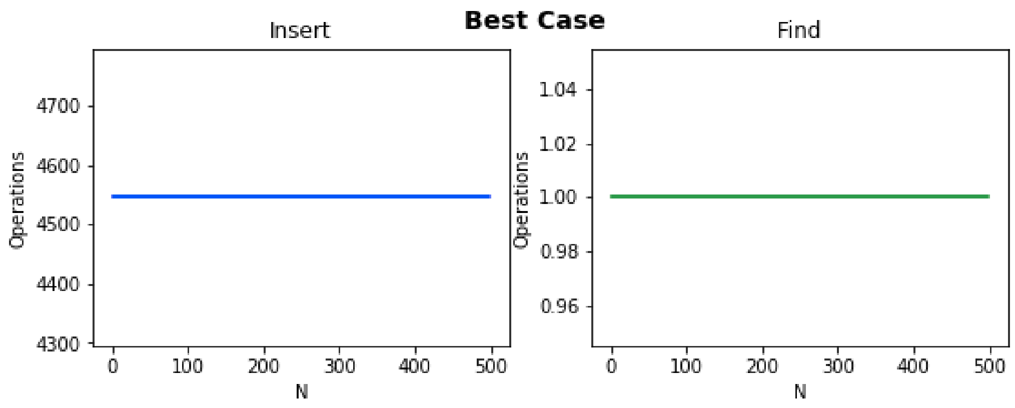

Figure 4: Best case for find and insert operation count as the data set increases in PowerBSTApp.

The best-case operation count for PowerBSTApp is constant because the first item to be searched for each set it always the first item in the binary search tree.

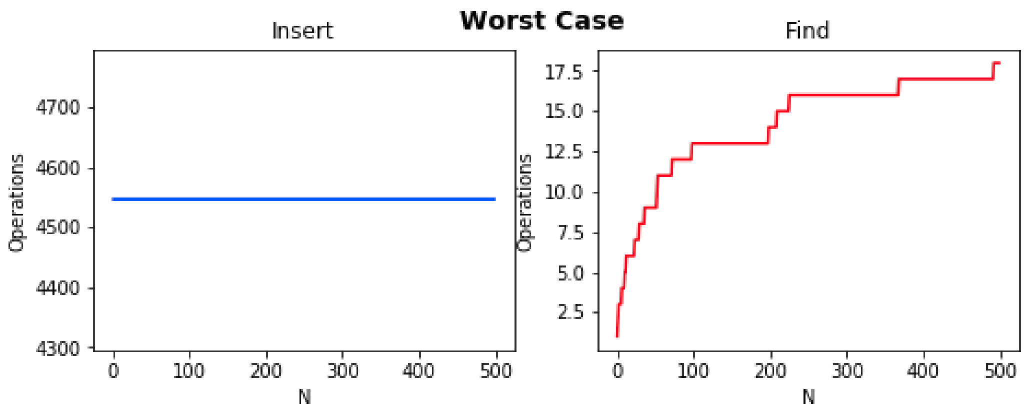

Figure 5: Worst case for find and insert operation count as the data set increases in PowerBSTApp.

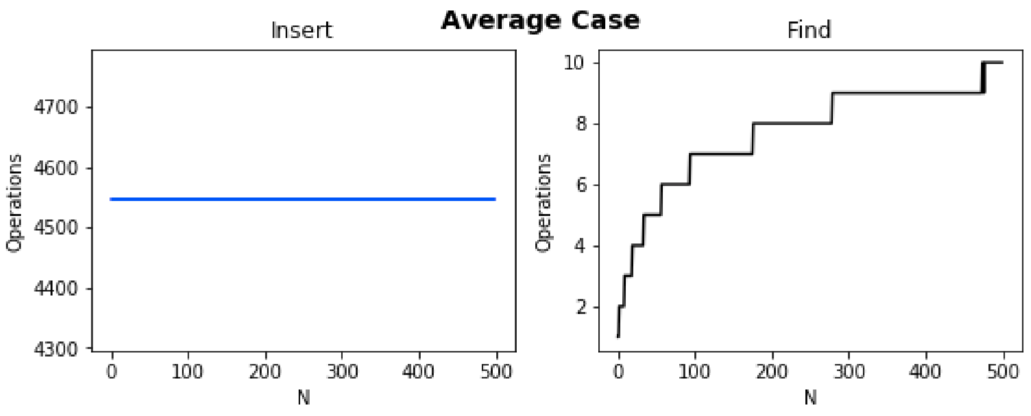

Figure 6: Average case for find and insert operation count as the data set increases in PowerBSTApp.

Similar to AVL the worst case takes on a logarithmic shape, where the operations increase at a slower rate than the value of N. It can also be observed that the final value for the worst case and average case is larger than for the AVLApp. Once again, this is due to the balanced property and therefore the height of the AVL is less than for BST.

Part 6

	Best Case	**Worst Case**	**Average Case**
AVL			
Find	1	9	7
Insert	3989	3989	3989
BST			
Find	1	500	250
Insert	124750	124750	127450

When creating a tree from sorted data, the two data structures behave differently. Binary search tree will create a linked list (one sided tree). AVL properties will not accept one side of the tree to be deeper by more than one and hence the tree will always be balanced.

From the values above, it can be observed that BST is O(N) for best, worst and average case. AVL has O(N) for best case and O($\log_2$N) for worst case.

The value for insert operations is constant for both application because of how the applications work. When run, they both create their respective trees once. AVL has a lower count for insert operations than Binary Search tree because its balance property ensured that one side of the tree is not deeper by more than one.

Conclusion

From the graphs and results above, it can be concluded that less “work” has to be done for AVL trees than binary search trees and therefore it is a more efficient data structure for adding and searching for data items.

Appendix

Git log

```
commit 5211a35d02361c4433e2cccd3b9879f8c91acbdc
Author: Niceta Nduku <NDKNIC001@myuct.ac.za>
Date: Sat Mar 16 13:34:29 2019 +0200

removed line in part 6 tests

commit 2c83fef6ae24b7e3d31eef3b3a852c42e75bbf81
Author: Niceta Nduku <NDKNIC001@myuct.ac.za>
Date: Sat Mar 16 13:13:27 2019 +0200

...
Author: Niceta Nduku <NDKNIC001@myuct.ac.za>
Date: Tue Mar 12 17:41:36 2019 +0200

Modifications made to source code

commit c19cd4abb89d27d7a9ca2128eaf06b8852867dcd
Author: Niceta Nduku <NDKNIC001@myuct.ac.za>
Date: Tue Mar 12 16:18:29 2019 +0200

Source code for Practical2
```